第3章

CPUの命令セットを拡張して高速化

PLD内蔵CPU「S5000」のアーキテクチャと活用事例

松添信宏，永峰 譲

ここでは，米国Stretch社のプロセッサ「S5000」を取り上げる．このプロセッサは，内部にプログラマブル・ロジックを搭載し，ユーザが命令セットを拡張できる特徴を持つ．ソフトウェアで時間を要する処理を，専用命令にすることで処理の高速化を実現する．本稿では，プロセッサの基本アーキテクチャを説明した後，画像処理アプリケーションにおける適用事例を紹介する．（編集部）

システム設計において，マイクロプロセッサは重要な要素の一つです．近年，システムの高機能化の要求が高まるに従って，マイクロプロセッサにも高い処理能力が求められるようになりました．そして，クロック周波数を上げるだけでは対応できないケースも増えています．

高い演算処理性能と制御性能が求められるシステムの設計手法の例を**図1**に示します．

まず，FPGA（field programmable gate array）にCPUコアを組み込むアプローチがあります．FPGAの中で，ステート・マシンや全体の制御用にCPUを積極的に利用します．処理性能が求められることが多い演算処理は，FPGAの論理ブロックを使ってハードウェア化します．

似たアプローチとして，CPUの内部にプログラマブル・ロジックを組み込んだ製品があります．単にCPUとプログラマブル・ロジックが1チップ化されただけではなく，CPUの命令セットをユーザが独自に拡張するしくみを備えています．プログラマブル・ロジックにより拡張命令を処理します．こうすることで，アプリケーションに適したオリジナルのマイクロプロセッサを設計できるようになります．

3

演算処理能力と制御性能の両立を目ざした製品としては，DSP（digital signal processing）命令付きRISCプロセッサやRISC命令付きDSPがあります．しかし，これらはメーカがあらかじめ用意した固定の命令セットを使うため，アプリケーションを選ぶ場合があります．

本稿で取り上げる米国Stretch社の「S5000」は，32ビットRISCプロセッサにプログラマブル・ロジックを内蔵したソフトウェア・コンフィギャラブル・プロセッサです．S5000プロセッサに内蔵されるプログラマブル・ロジックは，CPUの拡張命令を実行するためだけに使われます．これがCPUコアを内蔵したFPGAとS5000プロセッサの大



（a）プロセッサ内蔵FPGA　（b）プログラマブル・ロジック内蔵RISCプロセッサ　（c）DSP命令付きRISCプロセッサ

図1　演算処理能力と制御性能を両立させる手法

（a）の構成は，高い性能が得られる反面，LSI設計とソフトウェア開発の両方のスキルが求められる．（b）の構成では，プログラマブル・ロジックはCPUの拡張命令を処理するためにのみ使われる．このため，汎用プロセッサに近い感覚で使用できる．（c）の構成は，命令セットが決まっているので，性能が上がるかどうかはアプリケーションしだい．

KeyWord　S5000，FPGA，CPUコア，拡張命令，DSP，RISC，Stretch，IDE，ビルド，色空間変換，パイプライン

きな違いです．

内蔵のプログラマブル・ロジックのリソースが限られている以上，実現可能な性能ではFPGAを超えることはありません．汎用プロセッサに近い感覚で使うことができ，負荷の重い演算処理をハードウェア化するしくみを用意することで，性能向上と開発期間の短縮の両方を実現できることがメリットといえます(**図2**)．プログラマブル・ロジックの回路の記述はC言語をベースとする言語で記述ができるため，VHDLやVerilog HDLなどのハードウェア記述言語に関する知識が不要です．

1．S5000プロセッサのアーキテクチャ

S5000プロセッサの中核となるCPUブロック「S5エンジン」のブロック図を**図3**に示します．

ベースとなるCPUコアは，米国Tensilica社の「Xtensa-V」です．32ビットのRISCアーキテクチャで，最大300 MHzで動作します．命令セットは，16ビット長または24ビット長で構成されています．

S5000プロセッサには，搭載する周辺回路の違いによりいくつかの品種が存在しますが，S5エンジンそのものはすべて共通です．

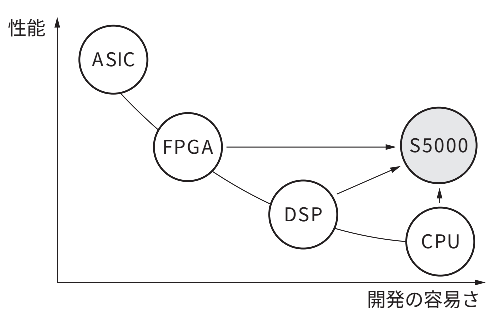


図2　性能と開発期間のトレードオフ

システム設計者はデバイスの性能以外に，アプリケーションの開発期間(TAT)を考慮する必要がある．この開発期間には所望の動作・性能を達成させるまでの最適化作業や検証・デバッグ期間が含まれる．S5000プロセッサは，汎用プロセッサ並みの開発期間でASIC/FPGAレベルの性能を出すことを目的としている．

● プログラマブル・ロジック(ISEF)

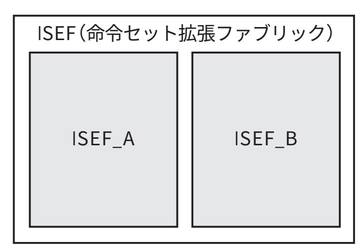
プログラマブル・ロジック部は，「ISEF(Instruction Set Extention Fabric；命令セット拡張ファブリック)」と呼ばれています．**図4**のように，ISEF_AとISEF_Bの二つのバンクで構成されています．

ISEFはユーザが設計したカスタム命令を実行するブロックです．Xtensaコアが発行する拡張命令の実行回路として使われます．

ISEFへのデータ入出力は，レジスタ経由で行います．ISEFへのデータ入力は，XtensaコアのARレジスタ(32ビット幅)から2系統と，ISEF専用のワイド・レジスタ(128ビット幅)から3系統あり，同時に3系統まで選択できます．ISEFからのデータ出力は128ビットが2系統です(**図5**)．

一つのISEFは，4,096個の算術/論理演算器(AU)と8,192個の乗算/シフト演算器(MU)，4,096ビット幅の拡張

図4
ISEFのブロック図
ISEF_AとISEF_Bの二つのバンクで構成される．



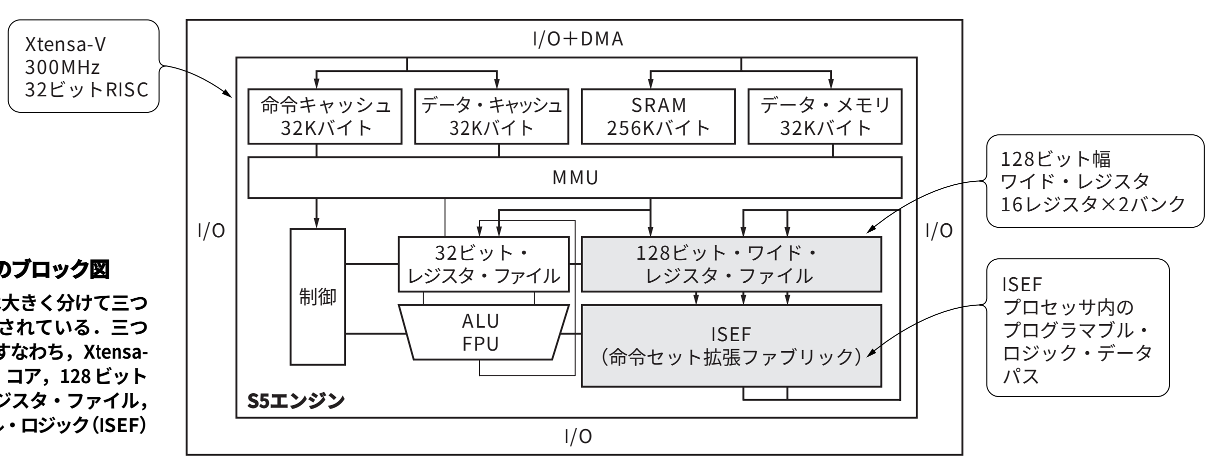


図3
S5エンジンのブロック図
S5エンジンは大きく分けて三つの要素で構成されている．三つの要素とは，すなわち，Xtensa-Vプロセッサ・コア，128ビットのワイド・レジスタ・ファイル，プログラマブル・ロジック(ISEF)である．

レジスタを持ちます．拡張レジスタは，演算結果の保持に使います．C演算子に対するリソース使用の目安は**表1**のようになります．一つのISEFで，最大32個の16ビット×16ビット乗算器または，128個の32ビットALU（あるいは最大256個の16ビットALU）を実装できます．

一つのISEFのコンフィグレーションで，最大16個の拡張命令を実装できます．そして，一つの実行ファイル（S5000のROMイメージ）に20セットのコンフィグレーションをリンクできます．

ISEFの書き換えは，

- オート・コンフィグレーション
- マニュアル・コンフィグレーション

の二つの方法があります．

オート・コンフィグレーションでは，拡張命令を実行するとき，その命令を持つコンフィグレーションがISEF_A，またはISEF_Bにない場合にかぎり，自動的に書き換えを行います．

マニュアル・コンフィグレーションは，拡張命令を使用する前にAPIを用いて，コンフィグレーションをプリロードします．これにより，書き換え時間のオーバヘッドを短縮できます．

ISEFの書き換えには，130 μs～150 μs程度の時間が必要です．ISEF_AとISEF_Bがともにコンフィグレーションされている場合，実行しようとする拡張命令に合わせてオーバヘッドなしで選択して実行できます．また，一方のコンフィグレーションでアプリケーションを実行している間に，もう一方を書き換えられます．

例えば，次のように四つの処理が別々のコンフィグレーションを必要とするとします．

- 処理1：Conf1にある拡張命令を使用
- 処理2：Conf2にある拡張命令を使用
- 処理3：Conf3にある拡張命令を使用
- 処理4：Conf4にある拡張命令を使用

二つのISEFバンクに対して四つのコンフィグレーションが必要となります．このような場合，**図6**のように，ある処理を実行中に次の処理で使用するコンフィグレーションをバックグラウンドで書き込んでおくことができます．

表1　C演算子に対するリソース使用の目安

C演算子	AU	MU
A(+, −)B	max(|A|, |B|)	0
A(&, ^, |)B	min(|A|, |B|)	0
A*B	0	|A|*|B|
A(<<, >>)B (Bは変数)	0	|A|*2|B|
A(<<, >>)B (Bは定数)	0	0

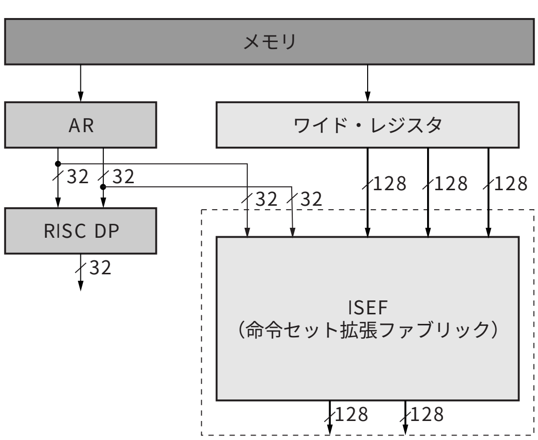


図5　ISEFのデータ入出力

ISEFへのデータ入力は，XtensaコアのARレジスタ（32ビット幅）から2系統と，ISEF専用のワイド・レジスタ（128ビット幅）から3系統あり，同時に3系統まで選択できる．ISEFからのデータ出力は128ビットが2系統．

● ワイド・レジスタ（WR）

ISEFへのデータ入出力専用には，128ビット幅の「ワイ

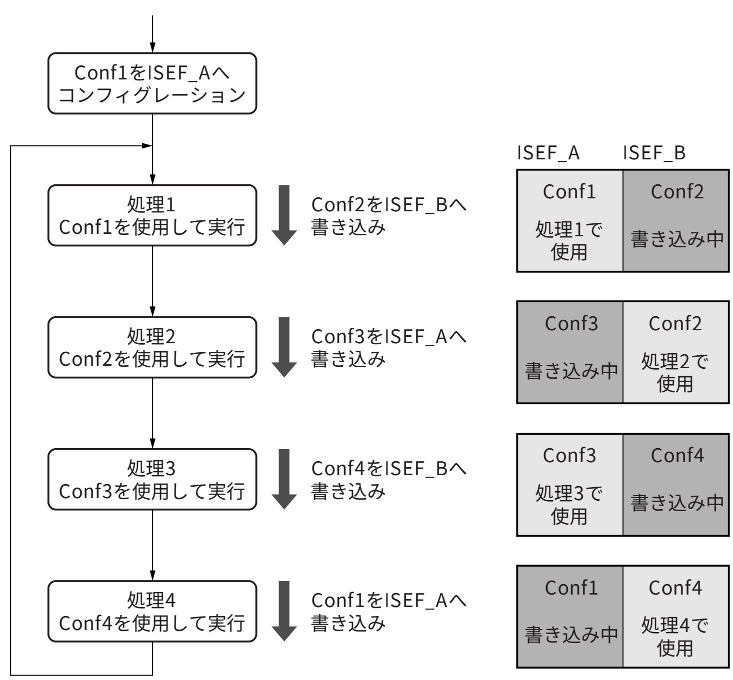


図6　二つのISEFバンクに対するコンフィグレーション

ある処理を実行中に次の処理で使用するコンフィグレーションをバックグラウンドで書き込んでおくことができる．

3

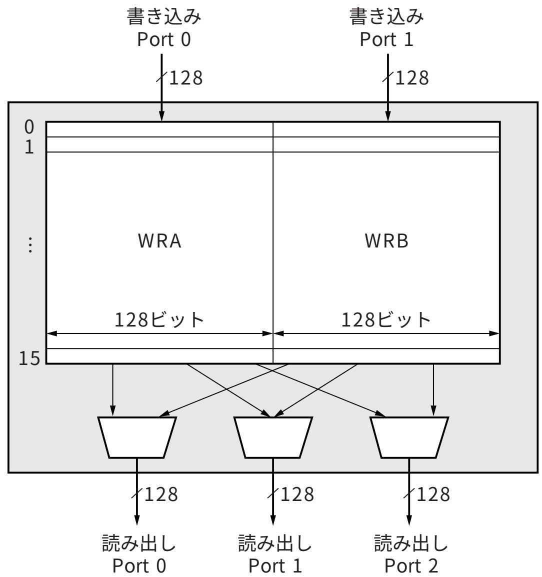


図7　ワイド・レジスタ（WR）の構成
WRAとWRBの2バンク構成になっている．

ド・レジスタ（WR）」を使用します．内部の300MHz，128ビット・バスへ接続されており，メモリ-ISEF間のデータのやりとりに必要なデータ転送帯域を持ちます．128ビット幅のため，8ビット長のデータであれば，16個のデータを1サイクルでロード/ストアできます．また，ピクセル当たり3バイトの画像データや12個の10ビット・データのような，データ幅が2のべき乗でないデータを扱えます．

ワイド・レジスタは，内部でWRAとWRBの2バンク構成になっています．WRAとWRBは，それぞれ16本のレジスタを持ちます．ワイド・レジスタからのデータ出力は，Port 0～Port 2まで3系統用意されており，出力ステージにあるデータ・セレクタで任意のワイド・レジスタ・バンクを選択します（**図7**）．

ワイド・レジスタとメモリのデータの受け渡しは，あらかじめ用意されているロード/ストア命令を使います．ロード/ストア命令は，最大128ビットのデータ・サイズに対

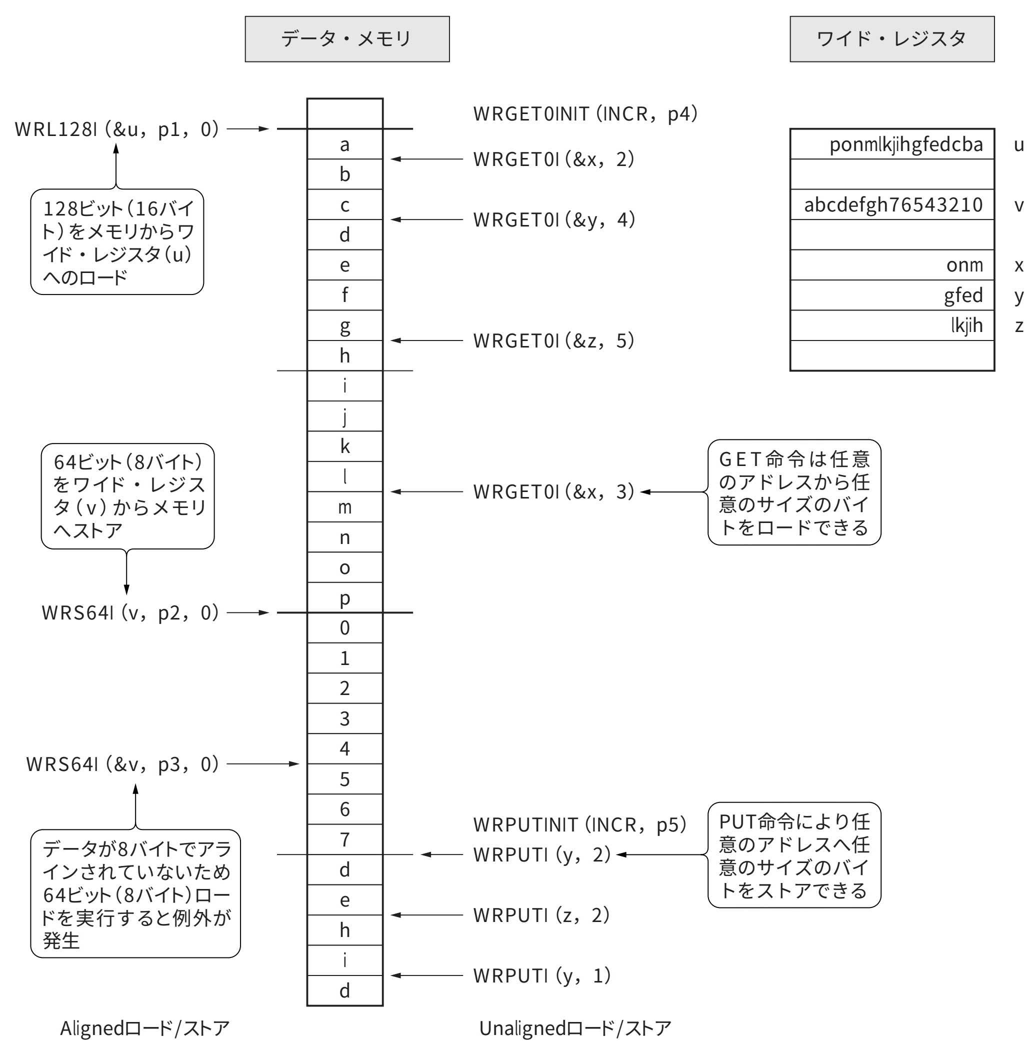


図8
ロード/ストア命令の動作
ロード/ストア命令には，Alignedロード/ストア命令とUnalignedロード/ストア命令があり，転送データの大きさによって使い分ける．Alignedロード/ストア命令を使う場合，*N*ビットのロード/ストア命令には*N*ビットでアラインされたアドレスでなければならない．画像データのような2の乗数ではないデータを扱う場合は，Unalignedロード/ストア命令が有効．

応します．また，アドレッシング・モードを選択できます．ロード/ストア命令には，Alignedロード/ストア命令とUnalignedロード/ストア命令があり，転送データの大きさによって使い分けます(**図8**)．**表2**にそれぞれの命令フォーマットを示します．

2．ソフトウェアの開発フロー

ソフトウェアの開発には，統合開発環境(IDE)の「Stretch IDE」を利用します(**図9**)．Stretch IDEは，プロセッサの実行コード生成だけでなく，ISEFのコンフィグレーション用ビットストリーム生成も行います(**図10**)．

IDEは，ソース・エディタ，Cコンパイラ，リンカ，デバッガ，プロファイラ，パイプライン・アナライザの機能で構成されます．

● IDEによるソフトウェア開発フロー

IDEによるソフトウェア開発フローの概要を**図11**に示します．ここではすでにRISCプロセッサで動作するC/C＋＋のアプリケーション・コードの開発が完了しているものとして説明を進めます．

アプリケーション・コードをコンパイル，およびビルド

表2
ロード/ストア命令

バンク	オペレーション	サイズ(ビット)	アドレッシング・モード
WR WRA WRB	L(ロード) S(ストア)	128 64 32 16 8	I(即値オフセット) IU(即値オフセット/ポスト・インクリメント) X(レジスタ・オフセット) XU(レジスタ・オフセット/ポスト・インクリメント) CU(サーキュラ・バッファ) RU(ビット・リバース)

(**a**) Alignedロード/ストア命令

バンク	オペレーション	ストリーム番号(GETのみ)	アドレッシング・モード
WR WRA WRB	GET(ロード) PUT(ストア)	0 1 2	I(即値バイト・カウンタ) X(レジスタ・バイト・カウンタ) XX(レジスタ定義個別バイト・カウンタ)

(**b**) Unalignedロード/ストア命令

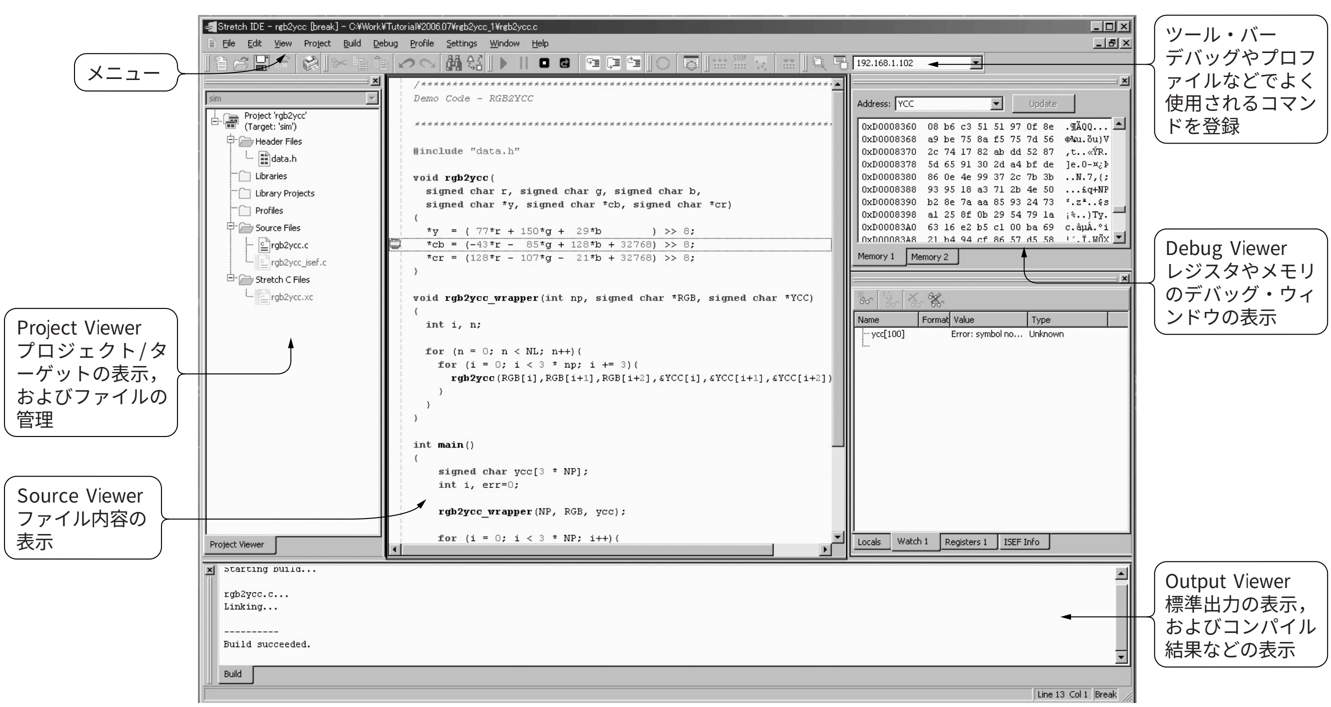


図9　統合開発環境「Stretch IDE」
Stretch IDEの画面を示す．ソース・コードおよびメモリ，プロセッサ・コアのレジスタ値を表示させている．

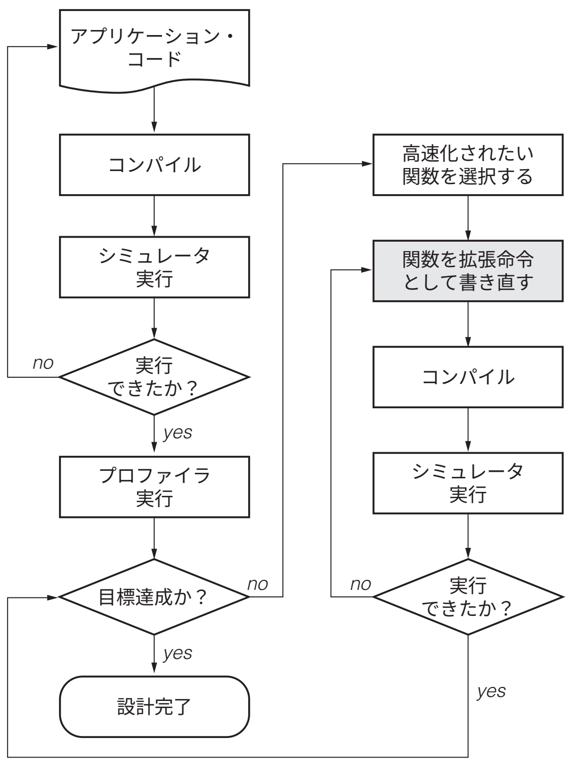


図11　ソフトウェア開発フローの概要
プロファイラの結果から，ボトルネックとなっている演算処理を見つけ，それを関数として切り出す．

し，プログラムが所望の動作をしているかを確認・検証します．ここまでは一般的なソフトウェア開発フローと同じです．

次に拡張命令を設計し，CPU負荷の重い処理をISEFにマッピングする作業を行います．CPU負荷を計測するツールとして，IDEにはプロファイラが用意されています．プロファイラの結果よりボトルネックとなっている演算処理を見つけ，それを関数として切り出します．そして，その関数をStretch-C言語に適合するようソース・コードに修正を加え，拡張命令にします．

● ビルド・プロセス

CPUが実行するアプリケーション・プログラムのソース・コードは，Cソース・ファイル（拡張子.c）に記述します．それに対しISEFで実行する拡張命令のソース・コードは，XCファイル（拡張子.xc）に記述します．

図12のように，Stretch-Cコンパイラ（SCC）は，拡張命令が書かれたXCファイルからコンパイルを開始します．そして，拡張命令をCソース・コード内で使えるように，ヘッダ・ファイル（libei.h）を生成します．同時にISEFコンフィグレーション用ビットストリーム・データ（libei.a）も生成します．

拡張命令はStretch-C言語で記述します．拡張命令のコンパイルの段階で，論理合成や配置配線，タイミング検証

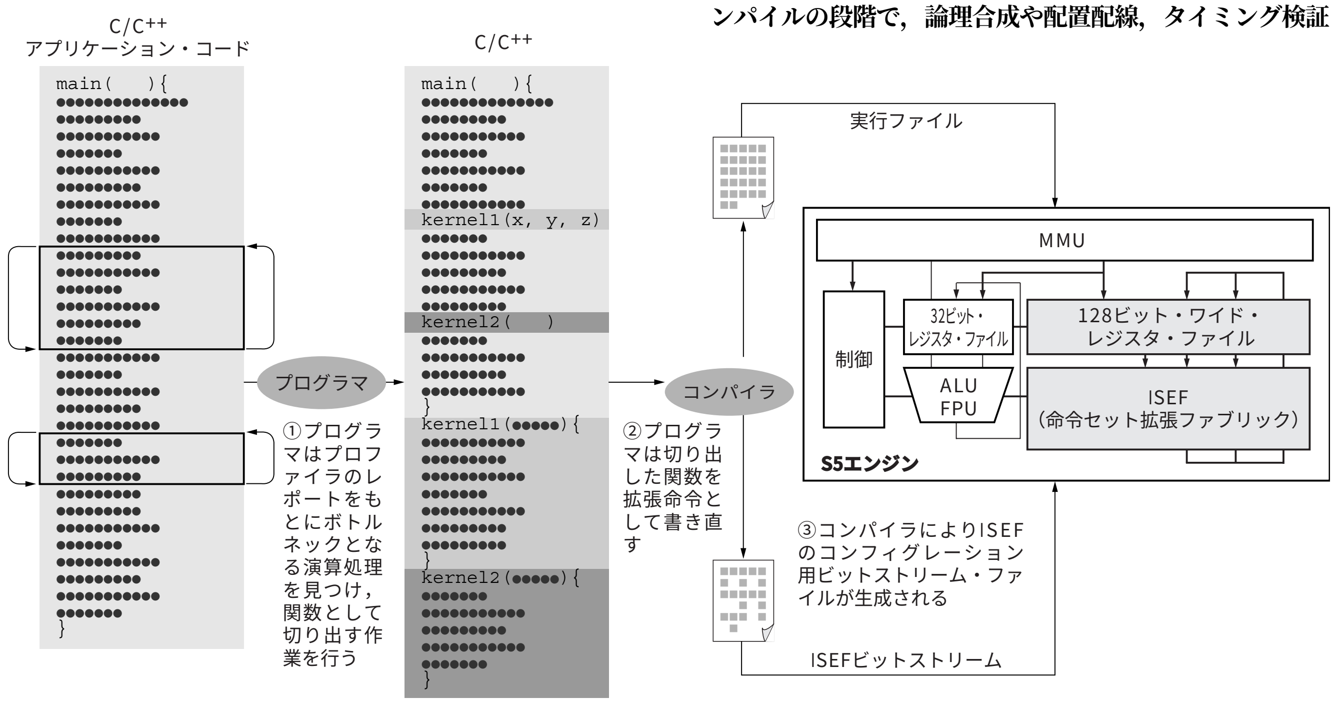


図10　IDEを使った開発の流れ
プロセッサの実行コード生成だけでなく，ISEFのコンフィグレーション用ビットストリーム生成も行う．

が自動で行われます．

次にCソース・ファイルをコンパイルします．リンカはISEFのコンフィグレーション・データとソフトウェアのオブジェクト・コードをリンクします．つまり，ISEFのコンフィグレーション・データは，実行ファイルに埋め込まれるため，特別な管理を行う必要はありません．

● 拡張命令を作る

拡張命令の設計には，C言語をベースとしたStretch-C言語を使用します．

拡張命令はSE_FUNC void型で定義します．関数名が拡張命令名となります．関数の引き数は入出力のレジスタを指定し，関数の本体ではプログラマブル・ロジックによる回路構成を定義します(**図13**)．

コンパイラはStretch-Cで定義された演算を実行するためのISEFコンフィグレーションを生成します．このとき，レイテンシを最小限(可能な限り並列化)にし，プログラマブル・ロジックの使用率を最小限[注1]にします．

Stretch-C言語でサポートされているデータ型を**表3**に示します．浮動小数点のデータ型はありません．これは，プログラマブル・ロジック内で浮動小数点演算をサポートをしないためです．

Stretch-C言語では，除算(/)とモジュロ(%；剰余)を除くC言語の演算子をサポートします．ただし，コンパイル時に値が決定されている場合には，除算とモジュロも使用できます．

また，ビット連結の演算子(**図14**)とビット抽出の演算子(**図15**)を拡張命令として使用できます．プロッセッサで実装されたこれらの演算は，マスク，シフト，AND，OR命令の組み合わせにより処理します．拡張命令はISEFで処理されるので，配線だけで実現できます．

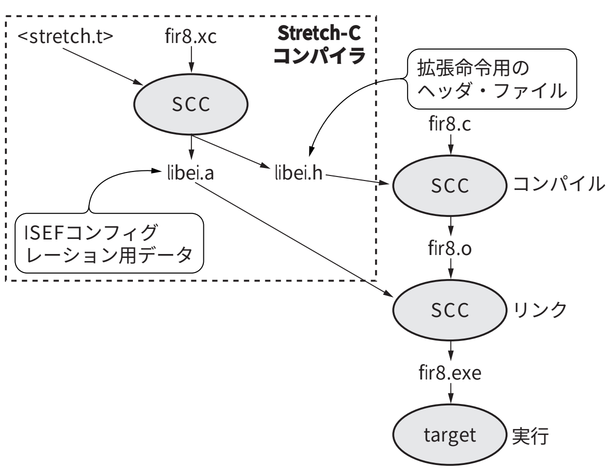


図12　ビルド・プロセス
ISEFのコンフィグレーション・データは，実行ファイルに埋め込まれる．

3．色空間変換を拡張命令で実現する

RGBからYCbCrへの色空間変換方程式を題材にし，色空間変換を拡張命令としてISEFへ実装し，アプリケーションの高速化を図ります．

● RGBからYCbCrへの色空間変換方程式

R，G，B，Y，Cb，Crのデータがそれぞれ1バイト(8ビット)のとき，固定小数点によるRGB色空間からYCbCr色空間への変換は，

注1：ISEFの最適化オプションはeffort level0-10で指定することができ，デフォルトは4を指定している．

```
SE_FUNC
void mul_add(WR xx, WR yy, WR *zz)
{
   short x = xx;
   short y = yy;
   int z = *zz;

   z = z + x * y;

   *zz = z;
}
```

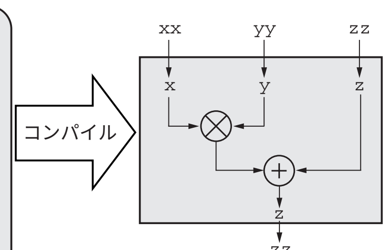


**図13
拡張命令の記述**
拡張命令はSE_FUNC void型で定義する．

**表3
Stretch-C言語でサポートされているデータ型**

標準整数データ型(符号付き/なし)	char，short，int，long，long　longなど
ワイド・レジスタ・データ型(128ビット符号なし)	WR，WRA，WRB
任意のサイズの整数データ型(符号付き/なし)	se_uint<m>，se_sint<n>(ただしn，mは任意の整数)

$$Y = (\ \ 77R + 150G + \ \ 29B) >> 8$$
$$Cb = (-43R - \ \ 85G + 128B + 32768) >> 8$$
$$Cr = (\ 128R - 107G - \ \ 21B + 32768) >> 8$$

となります．すなわち，1画素をRGBからYCbCrに変換する際に必要な演算は，8回の加減算，9回の乗算，3回のシフトになります．

リスト1は，C言語で記述した色空間変換プログラムです．関数rgb2yccは，色空間変換処理です． RGB入力データr，g，bを色空間変換し，YCbCrデータy，cb，crをポインタで渡します．関数rgb2ycc_wrapperは，処理を行う画素の数だけrgb2ycc関数を呼び出すラッパ関数です．

● 拡張命令の設計

拡張命令を設計する際には，以下の点に着目します．

- 複雑な演算を単一の拡張命令に実装する
- 128ビット幅のワイド・レジスタを効果的に利用する

ここでは，色空間変換方程式を単一命令に実装します．ワイド・レジスタを効果的に使うため，5画素を並列に処理できると考えられます．1画素のRGBデータ24ビット，YCbCrデータも24ビットです．したがって，128ビットのワイド・レジスタでは，5画素（120ビット）を一度にロード/ストアできます．つまり，「5並列の色空間変換命令を設計できる」と考えられます．

5並列色空間変換処理の拡張命令は，**リスト2**のようになります．

rgb2ycc関数は，1画素の色空間変換を行う関数です．入力は1画素分データでR，G，Bそれぞれ8ビットです．出力は1画素変換後のYCbCrデータです．符号なしの24ビット長にまとめるので，関数型はstatic se_uint<24>で定義します．色空間変換の式は，基本的にはオリジナルと同じものを使用できます．ここでは，**リスト2**の処理から，出力側をポインタから変数に変更しています．

rgb2ycc5関数は，5画素並列に色空間変換処理を行う拡張命令です．最初のforループでは，ワイド・レジスタAにパックされている5画素分のRGBデータを画素ごとにR，G，Bに分割します．次のforループでは，色空間変換関数rgb2yccを呼び出します．拡張命令では，for文

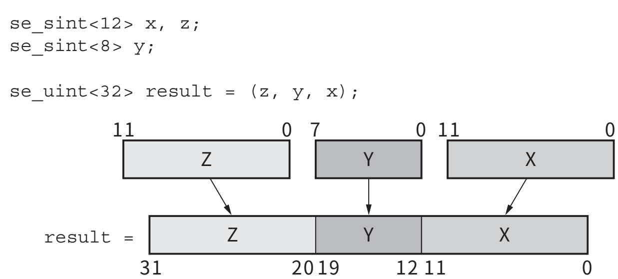


図14　ビット連結の演算子
result＝(z，y，x)；のとき，wのデータ長はx，y，zのデータ長の合計となる．

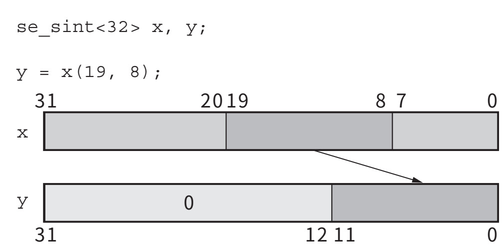


図15　ビット抽出の演算子
y＝x(19，8)；のとき，yにはxの8から19ビットまでがLSBから埋められ，MSB側は0で埋められる．

リスト1
色空間変換関数

```
void rgb2ycc(
  signed char r, signed char g, signed char b,
  signed char *y, signed char *cb, signed char *cr)
{
  *y  = ( 77*r + 150*g +  29*b        ) >> 8;
  *cb = (-43*r -  85*g + 128*b + 32768) >> 8;
  *cr = (128*r - 107*g -  21*b + 32768) >> 8;
}

void rgb2ycc_wrapper(signed char *RGB, signed char *YCC)
{
  int i, n;

  for (n = 0; n < NUM; n++ ) {
    for (i = 0; i < NP*3; i += 3) {
      rgb2ycc(RGB[i], RGB[i+1], RGB[i+2], &YCC[i], &YCC[i+1], &YCC[i+2]);
    }
  }
}
```

で記述していてもソフトウェア処理のように逐次処理されるわけではありません．コンパイラが依存関係を考慮し，並列処理可能なようにISEFへ実装します．

● アプリケーション・コードへの実装

拡張命令rgb2ycc5をアプリケーション・コード上に実装するには，データのロード/ストアも考慮する必要があります．5画素分のデータは120ビットのため，バイト境界を基準とするAlignedロード/ストア命令を使用することできません．**リスト3**のように，Unalignedロード/ストア命令のGET/PUTを使用し，rgb2ycc_wrapper関数を変更します．

まず，WRGET0INITとWRPUTINITにより，GET/PUT命令のストリーム・データをアクセスするためのアドレスを初期化します．

次のforループでは，

- WRGET0I命令でワイド・データAに15バイト分をロード
- rgb2ycc5命令で5並列色空間変換
- WRPUTI命令でワイド・レジスタBから15バイト分をストア

を行います．rgb2ycc5拡張命令は5並列の処理を行うため，forループの回数はオリジナルと比べて1/5になります．WRGET0I命令とWRPUTI命令は1サイクルで処理されます．

最後に，出力ストリームをメモリへフラッシュします．

リスト2　5並列色空間変換のための拡張命令

```
static se_uint<24>
rgb2ycc_1(se_sint<8> r, se_sint<8> g, se_sint<8> b)
{
  se_sint<8> y, cb, cr;

  y  = ( 77*r + 150*g +  29*b        ) >> 8;
  cb = (-43*r -  85*g + 128*b + 32768) >> 8;
  cr = (128*r - 107*g -  21*b + 32768) >> 8;

  return(cr, cb, y);
}

SE_FUNC void rgb2ycc5(WR A, WR *B)
{
  se_sint<8> r[5], g[5], b[5];
  se_uint<24> ycbcr[5];
  int i;

  for (i = 0; i < 5; i++) {
    r[i] = A(i*24+7,  i*24);
    g[i] = A(i*24+15, i*24+8);
    b[i] = A(i*24+23, i*24+16);
  }

  for (i = 0; i < 5; i++) {
      ycbcr[i] = rgb2ycc_1(r[i], g[i], b[i]);
  }

  *B = (ycbcr[4], ycbcr[3], ycbcr[2], ycbcr[1],
                                         ycbcr[0]);
}
```


● ISEFの使用リソースを調べる

拡張命令ファイル.xcをコンパイルすると，**図16**のようなレポート・ファイルが生成されます．

レポート・ファイルには，コンフィグレーション情報や拡張命令の情報，サイクル情報がレポートされます．リソース情報には，使用リソースの合計が，関数ごと，ソース・ラインごとにレポートされ，サイクル情報としては命令のレイテンシや拡張レジスタのレイテンシがレポートされます．

INSTRUCTION以下では，拡張命令の情報が提供されます．使用されているin，outとin/out変数（WRやAR）が「どの実行ステージ」で入出力されるか，使用されている拡張レジスタ（state）が「どの実行ステージ」で入出力されるかがレポートされます．また，拡張レジスタに関してはリード/ライト間のレイテンシ情報もレポートされます．**図16**では，拡張命令rgb2ycc5は，Aが入力，Bが出力で，実行ステージのサイクル1の始めでWRAから入力データを読み出し，実行ステージ2の終りでWRBへ出力データを書き込むことを示しています．したがって，rgb2ycc5拡張命令は2サイクル命令です（S5000コアを基準にすると6サイクル命令注2）．

Computational Resourcesは，コンパイル初期ステージに生成される見積り値です．5並列の色空間変換を実装す

注2：S5000コアは300MHzで動作するが，S5000のISEFは100MHzで動作するため．

リスト3　5並列色空間変換のための拡張命令を使ったアプリケーション・プログラム

```
void rgb2ycc_wrapper(signed char *RGB, signed char
*YCC)
{
  WR A, B;
  int i, n;

  for (n = 0; n < NUM; n++) {
    WRGET0INIT(0, RGB);
    WRPUTINIT(0, ycc);

    for (i = 0; i < NP/5; i++) {
      WRGET0I(&A, 15);
      rgb2ycc5(A, &B);
      WRPUTI(B, 15);
    }
    WRPUTFLUSH();
  }
}
```


3

図16
レポート・ファイル
実際にISEF上で使用されるリソースは，Final resource usage reportに示される．

```
// Total Number of configurations: 1
//
// Configuration 'rgb2ycc', implementing instructions rgb2ycc5:
//
//   INSTRUCTION=rgb2ycc5:rgb2ycc5 Opcode: 0: Assembly 'se_rgb2ycc5 A:in, B:out'
//     Input : variable 'A' read from WRA on ISEF read port 1 at beginning of cycle 1
//     Output: variable 'B' written to WRA on ISEF write port 0 at end of cycle 2
//
.....
//
//   Computational Resources
//     Arithmetic bits.................460
//     Logic bits.......................10
//     Mux bits..........................0
//     Register bits.....................0
//     Pipeline bits....................80
//   AU total.........................550 out of 4096
//     Multiply bits.................2240
//   MU total........................2240 out of 8192
//   Extension registers.................0 out of 4096
//
.....
/*****************************************************************************\
*                      Final resource usage report                              *
*---------------------------------------------------------------------------*
* Configuration rgb2ycc:
* Total AUs = 536 out of 4096
* Total MUs = 2240 out of 8192
* ISEF can run at the required frequency.
*
\*****************************************************************************/
```

コンフィグレーション数
コンフィグレーション名
見積もり値
ISEFの実際の使用数

る場合，550個のAU，2240個のMUを使用すると見積もられたことがわかります．

実際にISEF上で使用されるリソースは，Final resource usage reportに示されています．

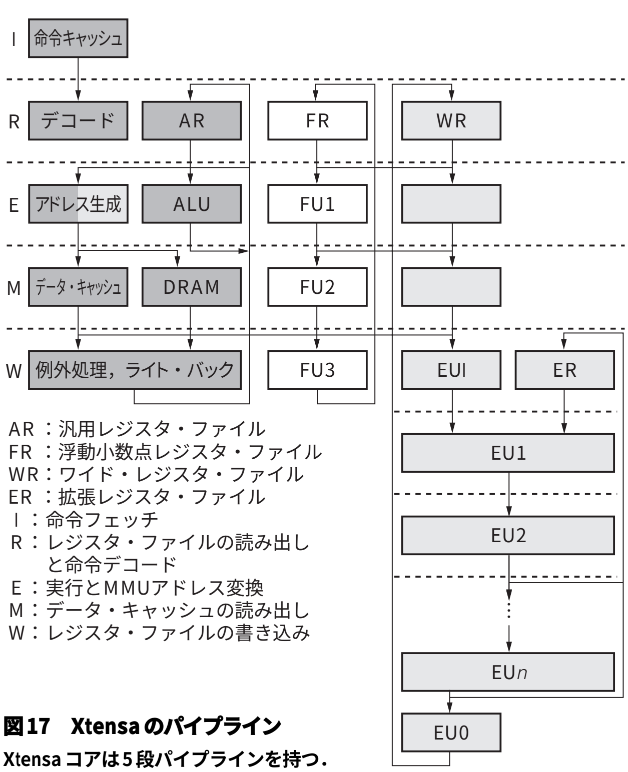


図17　Xtensaのパイプライン
Xtensaコアは5段パイプラインを持つ．

● パイプライン動作の解析

Xtensaコアは5段パイプラインを持ちます．拡張命令のパイプラインは，Xtensaのパイプラインを**図17**のように拡張しています．また，マルチサイクル命令をサポートし，最大31サイクルのレイテンシ（Xtesnaコア・クロック）まで持つことが可能です．実行ステージはEU1～EU27までをサポートします（拡張命令では9サイクル命令までをサポート）．

IDEには，**図18**のように，静的にパイプラインを解析する機能があります．データ依存によるストールの発生箇所を確認したり，パイプラインの動作を確認できます．拡張命令は，Xtensaコアの動作クロック300MHzに合わせて表示されます（ISEF側で2サイクルの拡張命令はXtensa側では6サイクル命令）．

S5000はインターロックのアーキテクチャを持っています．つまり，ある命令を実行するときに必要なレジスタ値（AR，WR，ER）をその前の命令が結果として出力する場合，命令実行の際に有効な結果が得られていないとハードウェアが自動的にストールを引き起こし，有効な値を待ちます．

例えば**リスト3**内側のループ（`for(i = 0; i < NP/5; i++)`）内において，`rgb2ycc5`からの出力Bは，10サイクルの間`WRPUTI`命令で使用できません（**図18**参照）．

図18
静的にパイプライン解析
データ依存によるストールの発生箇所を確認したり，パイプラインの動作を確認できる．縦軸は実行されたアセンブリ命令．横軸はXtensaクロック．WRPUTI命令はse_rgb2ycc5命令からの出力wra0の結果を待つため10サイクルをストールする．



図19
ループ展開実装後のプロファイル
1回のループ内に四つのse_rgb2ycc5命令を実行し，WRPUTI命令によるメモリへのストアのタイミングを遅らせることで，WRPUTI命令のストールをなくす．例えば，三つ目のアセンブリ命令se_rgb2ycc5の結果をストアするWRPUTI命令は13行目で実行．



図20
拡張命令実装前のプロファイル
64000画素の色空間変換に必要とされた実行サイクル数はrgb2yccとrgb2ycc_wrapperのTotalCyclesの合計3456826サイクル．

% Time	Total Cycles	Total Instrs	Calls	Cycles/call	I$ Misses	D$ Misses	Loads	Stores	Function
63.920377	2240057	2112000	64000	35	6	0	64000	192000	rgb2ycc
34.720694	1216769	960713	1	1216769	5	1	192402	4	rgb2ycc_wrapper

図21
拡張命令実装後のプロファイル
実装した関数rgb2ycc5は，プロファルではすでに命令として認識され，プロファイルには表示されない．64000画素の色空間変換に必要とされた実行サイクル数は41198サイクル（拡張命令の実装前と比較して約84倍）．

% Time	Total Cycles	Total Instrs	Calls	Cycles/call	I$ Misses	D$ Misses	Loads	Stores	Function
46.465831	41198	39407	1	41198	9	0	12201	12000	rgb2ycc_wrapper
23.188929	20560	4511	1	20560	13	3	3	0	ResetH
18.131577	16076	14897	1	16076	9	121	3844	0	main
3.691506	3273	2566	1	3273	3	0	0	0	xthal_dcache_all_writeback
2.572663	2281	1062	1	2281	8	70	12	263	_setup_page_tables
1.251931	1110	293	1	1110	60	20	66	63	_vfprintf_r

一方，命令間でデータの依存関係がなければ，パイプラインのストールを避けることができます．**図19**のように1回のループで四つの`rgb2ycc5`命令を実行し，結果をストアする（`WRPUTI`命令の実行）タイミングを遅らせ，パイプラインのストールを軽減できます．これをループ展開と呼びます注3．

● 性能の解析

命令セット・シミュレータを使用する場合は，関数ごとのプロファイルを生成できます．これをもとにして性能解析を行います．**図20**は拡張命令実装前のプロファイルを表します．**図21**は拡張命令実装後のプロファイルを表します．

実チップを使用した場合のプロファイル・データは**図22**のようになり，処理の割合のみが表示されます．しかしながら，実チップ上での性能解析は必須です．そこでStretch BIOSに組み込まれているPerformance Counterを使用します．これはsx-perfcount.hをインクルードして使用する

注3：ループ展開はデフォルトでコンパイラ・オプションとして実装されており，通常はループ展開したコードが生成される．

図22 実チップによるプロファイル
実チップでのプロファイルは全体の処理に対する各関数処理の割り合いのみが表示される．

%	Name	Graph
51.82	rgb2ycc_wrapper	
22.63	main	
4.38	_vfprintf_r	
1.82	sx_dmac_get_chan	
1.82	S_sx_load_isef_async	
1.46	sx_mmdma_chan_init	
1.46	sx_dmac_init_chan	
1.46	__sinit	
1.09	sx_virt_to_bus	
1.09	sx_mmdma_memcpy_init	
1.09	init_isef	
1.09	_malloc_r	

表4 Performance CounterのAPI

SX_COUT_STALLS_ALL	ストールの合計数を計測
SX_COUT_STALLS_ICACHE	命令キャッシュ・ミスによるストールの合計数
SX_COUT_STALLS_DCACHE	データ・キャッシュ・ミスによるストールの合計数
SX_COUT_STALLS_ISEF_ALL	ISEF命令コンフィグレーションによるストールの合計数
SX_COUT_INSN_ALL	命令の実行数
SX_COUT_INSN_LOADSTORE	ロード/ストア命令の実行数
SX_COUT_BRANCH_TAKEN	takenされた分岐命令の実行数

リスト4 実チップによる実行サイクル数/プロファイル情報の取得
実チップの性能解析はStretch社の用意するAPIを用いて行える．rgb2ycc_wrapper関数の実行処理サイクル数/プロファイル情報を取得．

```
sx_perf_reset(SX_COUNTER_0|SX_COUNTER_1);          ← パフォーマンス・カウンタをリセット
sx_perf_init(SX_COUNTER_0,SX_COUNT_INSN_ALL);      } 計測するイベントとカウンタを初期化
sx_perf_init(SX_COUNTER_1,SX_COUNT_STALLS_ALL);    }

sx_perf_enable(SX_COUNTER_0|SX_COUNTER_1);         ← イベントのカウント開始
cycle_beg = sx_get_ccount();                       ← フリー・ラン・カウンタの値の読み出し
rgb2ycc_wrapper(RGB, ycc);
cycle_end = sx_get_ccount();                       ← フリー・ラン・カウンタの値の読み出し
sx_perf_disable(SX_COUNTER_0|SX_COUNTER_1);        ← カウント終了

sx_perf_read (SX_COUNTER_0, &count0);              } イベントのカウント値の読み出し
sx_perf_read (SX_COUNTER_1, &count1);              }

printf("Cycle: %d\n", (cycle_end-cycle_beg));
printf("SX_COUNT_INSN_ALL        = %d\n",count0);
printf("SX_COUNT_STALLs_ALL      = %d\n",count1);
```

表5 S5530による色空間処理の性能比較

ターゲット	拡張命令	最適化	サイクル数	サイクル/1画素
ISS	なし	－O2	3456826	54.01
ISS	あり	－O2	41198	0.64
S5000	なし	－O2	3457160	54.02
S5000	あり	－O2	41393	0.65

APIを用いて，**表4**のイベントを計測するものです．ただし，同時に計測できるイベントは四つまでとなります．

一方，実行サイクル数の計測は，フリー・ランニングのタイマCCOUNTの値を測定することで算出できます．sx-timer.hをインクルードして使用するAPIで計測します．実チップの性能解析のためのコードを**リスト4**に示します．

表5にrgb2ycc_wrapper(NUM＝100，NP＝640)の処理サイクル数を，命令セット・シミュレータ(ISS)と**写真1**に示すS5000開発ボードで計測した結果，および1画素を処理(メモリのロード/ストアを含む)するために必要なサイクル数をまとめました．拡張命令を追加したことで約84倍の性能向上が見られました．

ISSとS5000とのサイクル数の違いは，ISSはプログラムを内部SRAMと同等のアクセス速度で実行しているのに対して，S5000はDDRメモリ上で実際に実行していることに起因していると考えられます．

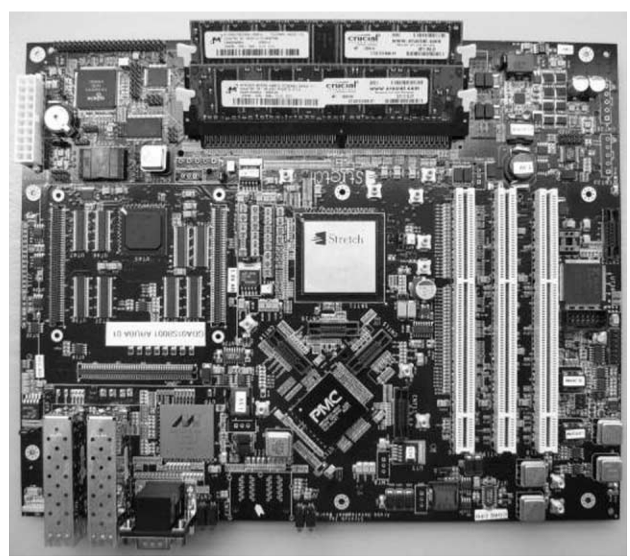


写真1 S5000開発ボード

参考・引用*文献

(1) Albert Wang，Cary Banta；ユーザ側でカスタム命令を追加できる汎用コンフィギャラブル・プロセッサを開発，Design Wave Magazine，2004年9月号.

まつぞえ・のぶひろ，ながみね・じょう
ストレッチ(株)